A Networking Subsystem for the Tera Operating System

Tera Computer Company
2815 Eastlake Ave E
Seattle, WA 98102

November 30, 1995

1 Introduction

This document proposes a framework and subsystem for networking in the Tera Operating System.

1.1 Requirements

The primary focus is to provide robust and timely TCP/IP functionality for the Tera. However, the design also accommodates the provision of NFS, OSI and ATM[1] in the future.

1.2 Goals

From a performance perspective, the challenge is for the networking subsystem to be ubiquitious and lightweight enough to 'make the hardware the bottleneck.'

To achieve this goal on the Tera's massively parallel architecture, this design attempts to minimize the following:

1. lock contention,
2. data movement,
3. checksum computation.

(2) and (3) are the traditional bottlenecks in uniprocessor TCP/IP implementations. However, the actual bottlenecks of network performance on the Tera system can only be identified after benchmarks are run on the actual system, and execution profiles examined.

[1] Asynchronous Transfer Mode

2 Choice of Code Base

Basically, this document proposes that the networking components of Unix 4.4 BSD be adapted for the Tera Operating System. Apart from Unix 4.4 BSD, the x-kernel and Parallel STREAMS were also considered.

2.1 Unix 4.4 BSD

A TCP/IP implementation based on Unix 4.x BSD will be widely interoperable. Many TCP/IP implementations are derived from Unix 4.x BSD. Consequently, its bugs and quirks of implementation have been preserved and widely propagated. It also incorporates significant public-domain contributions, such as Van Jacobson's optimization work.

The Unix 4.x BSD sockets library is a popular application programming interface for writing network programs. By using Unix 4.4 BSD as a code base, it will be easy to provide a sockets library for the Tera Operating System.

The Unix 4.4 BSD release also includes an NFS implementation and an OSI protocol stack. Using its TCP/IP stack will facilitate the integration of its NFS and OSI implementation into the Tera Operating System.

The primary disadvantage is that the Unix 4.4 BSD networking internals assume a non-preemptive, uniprocessor kernel. Hence, major modifications to the kernel portions of Unix 4.4 BSD for the Tera operating system will be necessary.

2.2 x-kernel Version 3.2

The x-kernel is an attractive platform for implementing network transport protocol engines. It is anticipated that research and experiments on transport protocols for high-performance gigabit networks will be implemented using the x-kernel. Hence, the Tera Operating System will eventually incorporate the x-kernel.

However, the current release of the x-kernel is not suitable as the initial code base for the Tera's networking subsystem, as significant effort will have to be spent to:

1. provide a sockets library and an interface for NFS. The current source release is either for a Mach server or a SunOS user-level process.
2. modify the x-kernel and the TCP/IP modules for the multi-threaded Tera Operating System. The current release assumes that the underlying environment is non-preemptive[2].

Also, it is suspected that the TCP/IP implementation provided with the current x-kernel release is not as feature-complete as that of Unix 4.4 BSD.

2.3 Parallel STREAMS

Parallel STREAMS is the framework for networking protocols in the Unix SVR4 ES/MP kernel The primary reason that it cannot be evaluated seriously is that its source code is not available.

[2]Modifications to rehost the x-kernel on a multiprocessor were discussed on the mailing list, xkernel@cs.arizona.edu. The discussions ranged from generalized queueing primitives to lock-free approaches to protocol implementation.

Part I

Functional Specification

3 Physical Interfaces

All network equipment connect to the Tera machine via HIPPI channels. To connect the Tera to a FDDI ring, the customer must acquire an IP router with at least a HIPPI and a FDDI interface. Similarly, to connect the Tera to an Ethernet segment, a T3 line, or a SONET/ATM network, the customer must acquire HIPPI/Ethernet, HIPPI/DS3 or HIPPI/ATM IP routers.

Figure 1 illustrates the anticipated network topology of a Tera installation.

3.1 IP and ARP over HIPPI

The Tera will transmit IP datagrams over a HIPPI channel as specified by RFC 1374. This ensures interconnectivity with other TCP/IP hosts and routers either via a point-to-point HIPPI connection or a network of HIPPI-SC switches. The PCHPI DX-HIPPI router from Network Systems Corporation complies with RFC 1374, as do routers from Computer Network Technologies.

To resolve an IP address to a HIPPI I-address over a HIPPI-SC network, RFC 1374 specifies three options:

1. manual configuration,
2. implementing ARP via an ARP server,
3. implementing ARP using multicast functionality.

Network Systems Corporation implements only option (1). Similarly, the Tera will provide only this option. Either the **arp** or **ifconfig** user-level commands will be used to manually associate a HIPPI I-address with an IP address.

This does not preclude implementing options (2) or (3) in the future, should customers demand it. No router vendor contacted so far has admitted to implementing options (2) or (3).

3.2 HIPPI Channel Allocation

RFC 1374 encourages short-lived HIPPI connections:

1. a source should limit its use of a HIPPI connection to 64 bursts[3]. This is at most 64 or 128 KB cumulatively.
2. one HIPPI packet[4] per HIPPI connection is acceptable,

[3] On a point-to-point HIPPI link between the Tera and an IP router, this may cause HIPPI connect/disconnect times to become significant. This will be reexamined after the initial implementation.

[4] Each HIPPI packet contains at most one IP datagram or datagram fragment.

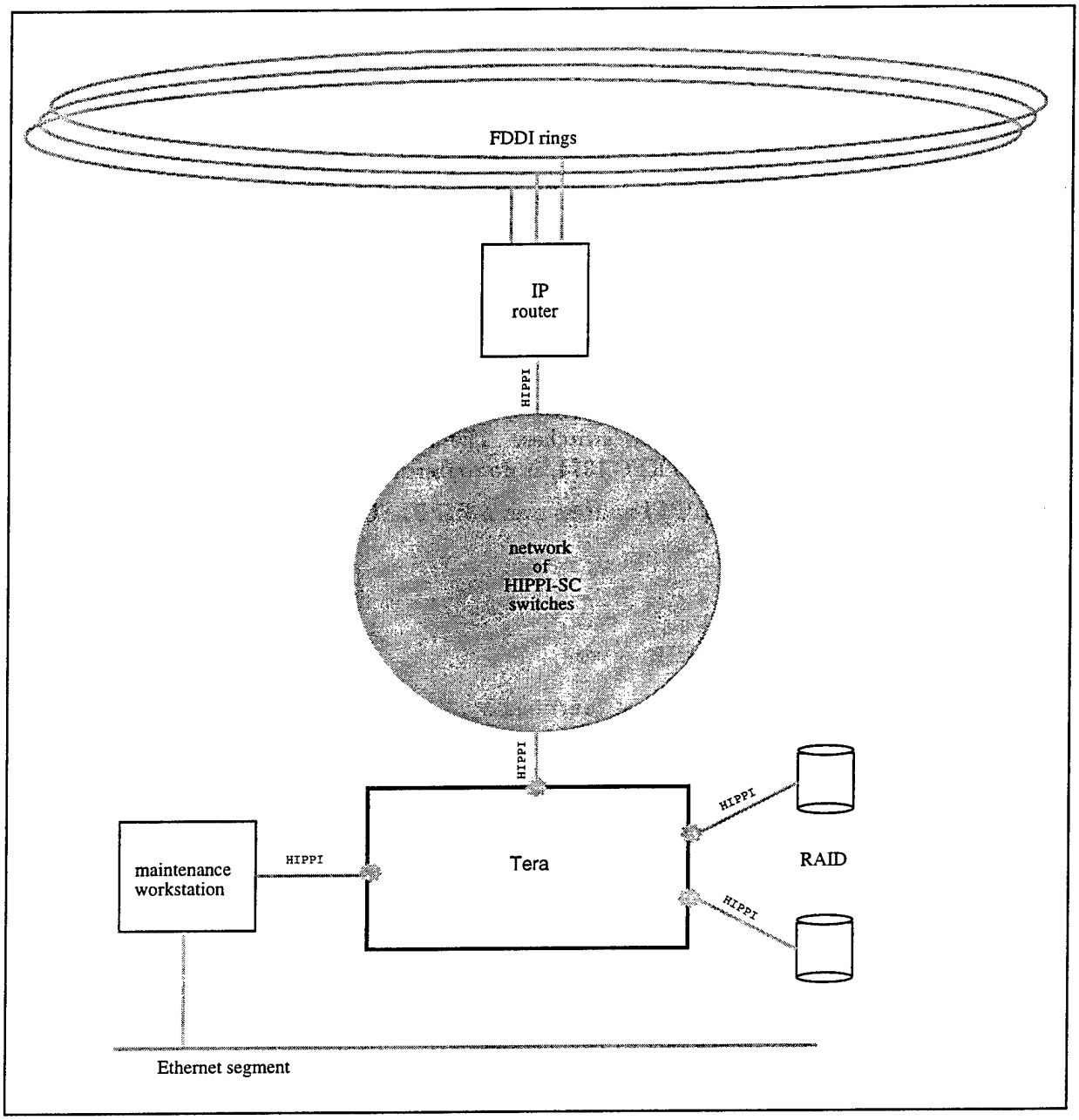

FIGURE 1: Physical Topology of a Tera Installation

However, a HIPPI channel utilized for disk i/o will probably transmit larger packets, since the HIPPI-FP specification allows HIPPI packets to be as big as $4Gb$. This implies long-lived HIPPI connections.

Hence, since disk i/o and network i/o differ as described, for performance reasons, a HIPPI channel should not be shared between the two. At least one HIPPI channel will be dedicated for network traffic.

Given the potential bandwidth of a HIPPI channel ($800Mbits\ s^{-1}$) relative to that of FDDI ($100Mbits\ s^{-1}$), it is anticipated that for most systems, a single HIPPI channel for network i/o will suffice. However, this design does not preclude dedicating multiple HIPPI channels for network i/o.

A minimal production Tera system will utilize the following HIPPI channels:

1. a point-to-point HIPPI link to the Sun console substation for the Tera console,
2. one or more HIPPI channels to RAID equipment,
3. a dedicated HIPPI channel for network i/o.

A Tera system has enough HIPPI channels to satisfy these allocations: even a 4 processor system will possess 4 HIPPI channels. . *new*

3.3 IP Interfaces

From a software perspective, the Tera has the following IP interfaces:

1. **lo0**, the software loopback device (IP address 127.0.0.1),
2. **hi0**, the HIPPI channel dedicated to network i/o,

4 User-Level Commands and Utilities

In this section, user-level TCP/IP utilities are partitioned into categories following the criteria established for Tera operating system utilities.

4.1 Class 2 (Early Prototype)

ftp	ping	rshd
ftpd	rcp	telnet
hostname	rlogin	telnetd
ifconfig	rlogind	tftp
inetd	route	tftpd
netstat	rsh	ttcp

The applications enumerated above must be available early in the life of the Tera hardware prototype. They enable remote access to the Tera system for system and compiler development.

4.2 Class 3 (First Customer Shipment)

arp	routed	talkd
mkfifo	ruptime	tcpdump
named	rwho	timed
named-xfer	rwhod	timedc
nslookup	syslogd	traceroute
rexecd	talk	trpt
rdist		

The TCP/IP utilities enumerated above are expected of a regular Unix TCP/IP release, and will be included in the Tera's First Customer Shipment.

4.3 Class 4 (After First Customer Shipment)

gated	htable	whois
gettable		

While routinely included in Unix releases, the TCP/IP utilities above can be provided after the First Customer Shipment.

4.4 Class 5 (Never)

XNSrouted	implogd	tn3270
implog[5]	slattach	

The Class 5 TCP/IP utilities are either obsolete or inappropriate for Tera systems. They will not be included in the Tera Operating System.

4.5 Deliberate Omissions

4.5.1 Yellow Pages and SUN RPC

domainname	rpc.rusersd	rusers
portmap	rpc.rwalld	spray
rpc.etherd	rpc.showfhd	traffic
rpc.lockd	rpc.sprayd	ypcat
rpc.mountd	rpc.statd	ypmatch
rpc.pwdauthd	rpc.yppasswdd	yppasswd
rpc.rexd	rpc.ypupdated	ypwhich
rpc.rquotad	rpcinfo	
rpc.rstatd	rup	

This document includes a plan to provide DNS functionality, but not Sun's Network Information Service, NIS, or NIS+. The primary obstacle is the lack of source code: Unix 4.4 BSD does not include them.

It is not clear whether Tera's customers will require NIS. Here at Tera, the lack of NIS on the Tera machine will make system administration inconvenient, but will not hinder its network availability.

4.5.2 Network File System

Although this design will allow the Unix 4.4 BSD implementation of NFS to be easily integrated, NFS is out of the scope of this document.

4.5.3 DCE and DME

This document does not include a plan for either the OSF Distributed Computing Environment or the OSF Distrbuted Management Environment. DCE includes Andrew, Kerberos and Apollo RPC.

4.5.4 Network Queueing System

This document does not include a plan to provide a network queueing system.

4.5.5 Mail

This document does not include a plan to provide an electronic mail package.

[5]The `tnamed` server, which supports the obsolete Name Server Protocol specified by IEN 116, will also not be supported.

4.5.6 Simple Network Management Protocol

The Unix 4.4 BSD TCP/IP implementation does not include SNMP functionality. It will have to be retrofitted in the future. This plan does not include that work.

4.5.7 Serial Line IP

Since the Tera machine will not have serial lines, a SLIP package will not be provided.

4.5.8 Reverse ARP

Reverse ARP is used mainly to boot diskless workstations. It relies on Ethernet broadcasts. The Tera will not need client-side RARP functionality. Also, since the Tera will not be connected directly to an Ethernet segment, there is no need for a RARP server on the Tera, and it will not be provided.

5 Programming Interfaces

5.1 User-Level Interfaces

The sockets interface is required by Tera's potential customers.

Sockets System Calls

accept()	read()	sendmsg()
bind()	readv()	setsockopt()
close()	recv()[6]	shutdown()
connect()	recvfrom()	socket()
getpeername()	recvmsg()	socketpair()
getsockname()	select()	write()
getsockopt()	send()[6]	writev()
listen()	sendto()	

Other Unix System Calls

gethostid()	getkerninfo(KINFO_RT)	sethostname()
gethostname()	sethostid()	

Macros

FD_CLR()	FD_SET()	FD_ZERO()
FD_ISSET()		

libc(3N)

dn_comp()	getservbyname()	ntohs()
dn_expand()	getservbyport()	rcmd()
endhostent()	getservent()	res_init()
endnetent()	htonl()	res_mkquery()
endprotoent()	htons()	res_send()
gethostbyaddr()	inet_addr()	rresvport()
gethostbyname()	inet_aton()	ruserok()
getnetbyaddr()	inet_lnaof()	sethostent()
getnetbyname()	inet_makeaddr()	setnetent()
getnetent()	inet_netof()	setprotoent()
getprotobyname()	inet_network()	setservent()
getprotobynumber()	inet_ntoa()	
getprotoent()	ntohl()	

[6] Although documented as system calls, send() and recv() are actually library functions implemented with sendto() and recvfrom() in Unix 4.4 BSD.

libresolv.a

`gethostbyaddr()`	`res_mkquery()`	`res_search()`
`gethostbyname()`	`res_query()`	`res_send()`

libcompat.a

`rexec()`[7]

5.1.1 Multi-Threaded Considerations

`gethostbyaddr()`	`getnetent()`	`getservbyport()`
`gethostbyname()`	`getprotobyname()`	`getservent()`
`gethostent()`	`getprotobynumber()`	`inet_makeaddr()`
`getnetbyaddr()`	`getprotoent()`	`inet_ntoa()`
`getnetbyname()`	`getservbyname()`	

The libc(3N) functions enumerated above are neither reentrant nor thread-safe: as defined by Unix 4.4 BSD, they return pointers to static data. Thread-safe variants of the functions will be be defined and implemented.

5.2 Supervisor-Level Interfaces

The networking components that reside in the Unix 4.4 BSD kernel are organized into three logical layers:

- the socket layer,
- the protocol layer,
- the network interface layer.

So that Unix kernel-level components such as NFS and other protocol stacks can be ported over to the Tera, the following networking interfaces will be available:

Socket Layer Support Routines

`soaccept()`	`socreate()`	`sosend()`
`sobind()`	`sogetopt()`	`sosetopt()`
`soconnect()`	`solisten()`	`soshutdown()`
`soconnect2()`	`soreceive()`	

[7] In Unix 4.4 BSD, this has been made obsolete by `krcmd(3)`.

mbuf Management Routines

MCHTYPE()	M_PREPEND()	m_get()
MEXGET()[8]	M_TRAILINGSPACE()	m_getclr()
MFREE()	dtom()	m_gethdr()
MGET()	m_adj()	m_prepend()
MGETHDR()	m_cat()	m_pullup()
MH_ALIGN()	m_copydata()	m_split()
M_ALIGN()	m_copym()	mtod()
M_COPY_PKTHDR()	m_free()	
M_LEADINGSPACE()	m_freem()	

Network Interface Layer Routines

IF_ENQUEUE()	IF_DEQUEUEIF()	IF_PREPEND()
IF_DEQUEUE()		

[7] MEXGET() replaces MCLGET().

6 Future Enhancements

The following features are missing from the Unix 4.4 BSD pre-alpha release, and will be added.

6.1 TCP/IP Extensions

6.1.1 High Performance TCP

In the Unix 4.4 BSD pre-alpha TCP implementation, the maximum TCP segment size is 64 KB. RFC 1323 specifies a TCP window scale option that allows TCP segments to be 2 GB. This may be necessary to achieve good TCP throughput over a HIPPI channel.

RFC 1323 also specifies a method to timestamp each individual TCP segment, so that the round-trip-time of a TCP connection can be accurately measured.

6.1.2 MTU Discovery

Media	*Maximum Transfer Unit*
HIPPI	64 KB
ATM AAL5	9180 bytes
FDDI	4352 bytes[8]
Ethernet	1514 bytes

Since the Maximum Transfer Unit, or MTU, for a HIPPI channel is much larger than the MTU size of FDDI and ethernet, MTU discovery as specified by RFCs 1435 and 1191 should be implemented so that it is possible to send optimal sized TCP segments across IP routers.

6.1.3 IP Multicasting

IP multicasting, as specified by RFC 1112, may be necessary to support multimedia applications.

6.2 Buffer Management

6.2.1 Extended Mbufs

Right now, `M_EXGET` fails if there isn't sufficient memory for an mbuf extension of the requested size. This needs to be refined so that the caller calls `M_EXGET` to get 2 buffers of half the size, ... and so on.

[8] page 5, RFC 1390, January 1993.

Part II
Design and Implementation

7 Processing Activity

Activity[9] within the network subsystem begins when:

1. data is sent down a socket,
2. a packet is received from the HIPPI channel,
3. a function within the network subsystem is fired because its `timeout()` period expired.

7.1 Levels and Protection Domains

All network processing executes at supervisor level (`LEV_SUPER`), either in:

1. the protection domain of the task that originated the activity,
2. the kernel/supervisor daemon's protection domain.

(1) is desirable because it preserves the 'parallelism width' of a task, and avoids the overhead of inter-address space communication. However, the process scheduler will not pm-swap a task while it has chores executing at supervisor level.

Hence, to prevent adverse interaction with the process scheduler, an SP-chore will execute in the network subsystem only for its own benefit. It will avoid processing network packets that are not destined for it.

7.2 Parallelism and Load Balancing

For efficiency, throughput and scalability, the networking subsystem will process as many network packets in parallel as possible. For outbound network packets, most of this parallelism will be achieved by relegating network processing to the originating chore, as described in section 7.1.

Parallelizing the processing of inbound network packets is more problematic. As shown in figure 2, the Unix 4.4 BSD TCP/IP implementation processes network packets in layers. The ultimate task/chore destined to receive the packet, if any, cannot be determined until the TCP header of an inbound packet is analyzed, and associated with a TCP control block.

Hence, to process inbound network packets in parallel, it is necessary to have concurrent activity within the kernel/supervisor daemon's protection domain. However the 'parallelism width' dedicated to networking there must satisfy the following constraints:

[9] The `accept()`, `bind()`, `listen()`, ... system calls are ignored here.

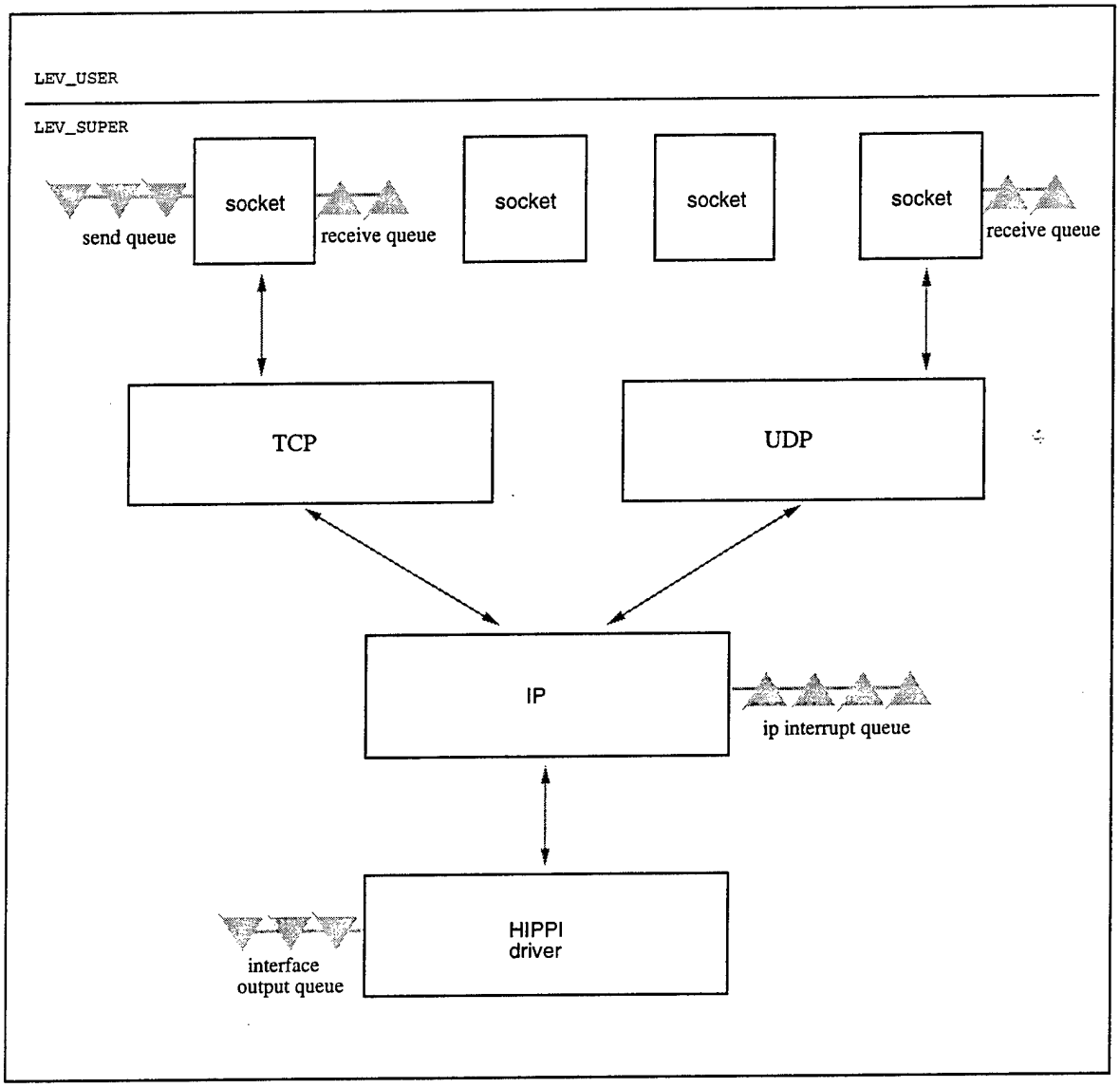

FIGURE 2: TCP/IP Processing

1. there should not be so many hardware streams processing network packets that the parallelism of user-level tasks is affected. This occurs if the user-level scheduler is deprived of hardware streams.

2. there should not be so few hardware streams processing network packets that user-level chores blocked in a `read()` waiting for data to percolate from the hardware interface up to the socket incur unnecessary delays.

However, when the bandwidth of a HIPPI channel is considered relative to the processing speed of a hardware stream, it may prove in practice that only a few daemon chores are necessary to service many user-level chores waiting within `read()`.

7.3 Using a 'Network Processing Server'

new

To facilitate a dynamic 'parallelism width' for processing inbound network packets, it is tempting to relegate such processing to a server task. This task would consist only of SPchores, all executing or `sleep()`ing within the network subsystem. When parallelism needs to be widened, more chores are created within this task; when parallelism needs to be shrunk, some chores are destroyed.

The problem with this scheme is that the server task will consume one protection domain on each physical processor. Protection domains are treasured resources, there being only 16 per physical processor.

However, one protection domain on each physical processor is already reserved for the kernel/supervisor daemon. Hence, this design proposes that inbound network packets and other such network processing take place in the kernel/supervisor protection domain.

7.4 timeout() Parallelism

The Unix 4.4 BSD kernel networking subsystem uses `timeout()` to implement the following timers[10]:

1. a fast protocol timer, which fires 5 times a second (once every 200 ms),

2. a slow protocol timer, which fires twice a second (once every 500 ms),

3. the arp cache timer, which fires every 5 minutes.

`ip_slowtimo()` discards old IP fragments waiting for reassembly. It is not anticipated that parallelizing `ip_slowtimo()` or the arp cache timer will contribute to any speedup.

`tcp_fasttimo()` sends down delayed ACKs for those TCP connections that have acknowledgments pending; `tcp_slowtimo()` retransmits TCP segments, probes windows, and sends keepalive heartbeats on those TCP connections that need it. These two TCP timers will be modified to process

[10]The networking subsystem also includes the interface watchdog timer, which fires once a second. However, the HIPPI driver will probably not use this timer.

individual TCP control blocks concurrently.

```
/*
 * slow protocol timer scheduler
 */
pfslowtimo()
{
    create a chore to call ip_slowtimo();
    create a chore to call tcp_slowtimo();
    timeout(pfslowtimo, (caddr_t) 0, hz/5);
}

tcp_slowtimo()
{
    register struct tcpcb *tp;

    lock the linked list of TCP control blocks;

    for (tp = head_of_list; tp; tp = tp->next_in_list) {
        lock the TCP control block;

        if (this TCP connection needs timeout processing) {
            increment reference count of the TCP control block;
            unlock the TCP control block;
            create a chore to send a packet down the TCP connection;
        }
        else {
            unlock the TCP control block;
        }
    }

    unlock the linked list of TCP control blocks;
}
```